On the Development of Open Source System Software for High Performance Computing

Bill Saphir
February, 2000
wcsaphir@lbl.gov

Background

This position paper is written in the context of the HPC Open Source working group, which first met in Santa Monica in August 1999. It is intended to complement the output of this working group. The working group has focused on technical issues, assessing the state of the art and requirements three to five years in the future. The written output of the group has paid less attention to carefully defining both the motivation and ultimate goals, and to assessing alternatives for "process" as well as "product". To some extent, this lack of attention is the intentional result of trying to avoid sounding too much like a proposal, but it may also mask difficult or contentious issues.

The premise of this paper is that it is better to discuss motivations and goals early in the process, in order to ensure that the working group is in fact operating on a single wavelength, and to alert decision makers to areas that will need careful attention later on. The intent of this paper is to capture some of the ideas that have been discussed, and provide a reference point for further discussion.

Summary

- There is a looming crisis for high performance computing (HPC): proprietary systems may not be able to fill HPC needs 3 to 5 years from now.
- Cluster system software development is not on track to fill these needs.
- The HPC community (especially government) cannot rely on market forces to fill the gap and should take steps to ensure the continued viability of high-end systems.
- We must not underestimate the degree of robustness and functionality needed, or the development effort needed to provide these.
- Open Source software is a necessary part of the solution. Open source will help to ensure that we do not end up in the same state we are in today.
- Open Source does not by itself ensure success – coordination/organization is chief difficulty
- Key questions: How can we make the coordination happen? What should be the interaction between government and vendors?

1.1 A Looming Crisis for HPC

High performance scientific computing has become critical to scientific research and to meeting mission-critical requirements of government agencies. As the problems that must be solved become more complex and their analysis more sophisticated, ever-larger computing resources are needed.

In its infancy, scientific computing was a driving influence in the computer market. As the computer industry has grown, scientific computing has grown also, but not as quickly, so that it is now a niche market in the overall economic picture of the computer industry. Rather than driving the market, scientific computing now largely rides on its coattails, taking advantage of the dramatic increases in speed and capacity of mass-market hardware.

While scientific computing can often[1] leverage mass-market hardware, the mass market does not need complex system software (operating system) run by high-end systems. There is commercial market for

[1] Mass-market hardware is not always good enough. For instance, some scientific problems require very low latency networks that aren't being developed. ASCI Pathforward has had some success in driving the development of these networks, but it has not entirely succeeded. It is unlikely that new industry initiatives

moderately sized parallel computers that function as back-end servers for commercial applications such as databases and web servers. Unfortunately, system software developed to support these applications does not meet the needs of high performance computing, and may not scale to extremely large systems. High performance computing has therefore become a niche market with large development costs. Commercial viability has been sustained to date more by the cachet (PR value) of fielding the fastest computer than by revenue stream.

The toll on the high performance computing world has been large, and the HPC community is concerned that the problem will only grow worse[2]. Software development costs have been a major factor in the demise of several high performance computing failures. Intel and Cray Research are also no longer building distributed memory parallel computers based on commodity technology. The Cray T3E, whose system software is considered by many to be the most full-featured and robust software in the industry, is a dead-end product, and its software is unlikely to be used elsewhere[3]. Sun and Digital/Compaq have struggled for years to develop credible system software at the high end, and have not yet succeeded. HP gave up early on and now focuses, with much success, on moderate-size systems. SGI (the old "Irix Forever" SGI, that is) has also stopped at moderate sized systems (512) and not had much success with larger systems (e.g. ASCI Blue Mountain). Very visible cluster efforts by a number of Linux startup companies are focused on high availability rather than high performance, or are not yet credible as complete HPC solutions for very large clusters. Only IBM has demonstrated usable system software for very large computers that will continue to be available for the foreseeable future. In the opinion of many, this software still has a long way to go, having shown only incremental improvement over the past 6 years, and is missing important features that are available on other systems.

The situation today, while not perfect, is tolerable. In this paper we are concerned with what will happen 3 to 5 years from now. As the size of high-end computers continues to grow and expectations increase, it will become less and less feasible for companies to provide robust system software. Among other reasons, no company is likely to have a 10,000-node system on which to test software. Moreover, a number of promising systems, including some associated with the Petaflops initiative, will face a software crisis. There is no indication that industry will step in and save the day.

To summarize, two facts endanger the continued viability of HPC system software:

- The market for HPC system software, which is complex and expensive to develop, is not economically viable.
- The considerable effort that is being put in to development is split many ways and therefore diluted.

To ensure that HPC needs continue to be met, the HPC community must take steps to ensure the continued availability of high-end systems, leading industry rather than waiting for commercial solutions. A recommendation for how to do this is the topic of the rest of this paper.

1.2 Production Software

A number of unstated assumptions about goals for system software "quality" underlie the latest HPC OS WG document (the "Santa Fe" document). We capture some of them here so that they are explicit. This is both to help ensure that everyone is in agreement on the goals and so that we don't forget about them later on.

---

such as InfiniBand will significantly improve matters. In this position paper, we do not address the hardware problem, other than to acknowledge that the mass market does not have a complete solution.

[2] In this paper, we do not consider Japanese computers, at least two of which appear to provide very high performance in a production environment. The first reason is that it is not politically feasible to consider them. Until Washington gives a clear indication it is willing to consider foreign-made supercomputers, they are moot. Should the first reason evaporate, it is still not at all clear that these computers could step in to fill the gap, as their price-performance is low, and vector architectures are no longer suited to the majority of US codes, which have now been optimized for microprocessors with hierarchical memory systems.

[3] Some optimists believe that T3E software will resurface in the Cray SV2 product.

1.2.1 Don't lower the bar

We take a moment to define ***production*** software. Because the largest machines available cost tens of millions of dollars, it is critical that they be used effectively.

The most important feature of production software is that it must enable high utilization[4] (cycles delivered to an application as a fraction of available cycles at 24 hours per day 365 days per year). The trick is to achieve this while implementing policies that can be difficult to implement, such as quick turnaround for debug and high priority jobs. Sophisticated features such as gang scheduling and checkpoint/restart make it easier to achieve high utilization and responsiveness (and recover from failures)[5]. To achieve high utilization, it must also be possible to quickly install, test and de-install new software, predict, prevent, detect and recover from failures. It must be possible to prevent failures due to users filling up the disk, logging into nodes they shouldn't, and other accidental asocial behavior. Administrators must be able to keep things running well and still get sleep and spend time with family.

A production environment has tools that help users develop and tune applications, a high performance and reliable I/O system providing global storage, good compilers, and other things to make life easier for users. It has robust accounting needed both to satisfy the people paying the bills and to tune the system. Not too many years ago, it was standard on HPC systems to record floating rates of all applications and number of bytes of I/O.

Not all of these things are available on the best of today's systems, and we can all tell self-pitying stories of how it was better in the good ol' days. The point here is that in some ways we are willing to accept less and less for system software in an era when most software is providing more and more.

1.2.2 Keep standards in place

Later in this document we will argue that the HPC community should develop its own software. In the current model, where we purchase large machines from vendors who also provide the software, we hold vendors to high standards[6]. In order for a large machine to be accepted, it has to pass a series of functionality and performance tests essentially demonstrating that it has a production environment and performs as advertised.

If we are developing our own software, it is no longer us against them; there is no longer a bad guy whom we can blame when things go wrong. There are two ways we could react to this. One is to avoid rigorous standards, letting them slip when the going gets tough; the other is to continue to use these standards to rigorously and scientifically assess where we are. The temptation to declare success and move on makes it tough to pursue this course.

Clearly not all sites will need all features described above. Some will have minimal accounting requirements; some will run only one application; some will not even care about MPI. If we want to have a community software base, our system software must meet more of the needs of more HPC centers, and the infrastructure should facilitate adding of functionality by sites that have specific special-case requirements.

---

[4] Apparently a few large machines do not have a requirement for high utilization, as their purpose is to provide quick turnaround on occasional critical large jobs. Implicit in such a policy is the realization that job scheduling software is not capable of preempting/suspending low priority jobs, among other deficiencies. If this technical problem were solved, it would insane not to fully utilize such a machine.

[5] The use of gang scheduling and checkpoint/restart to improve utilization and responsiveness is not really in doubt – the debate about them ultimately boils down to whether the extra cycles gained are worth the development and hardware costs, which might be huge.

[6] Well, usually. Standards seem to have been slipping to the point where some very expensive systems could hardly be called production.

1.2.3 Don't underestimate the work required

There has been a tendency in discussions of system software to declare that because a solution for X exists at site A, X has been solved. In fact, this is no more than a proof of concept. Getting from the original solution to a full solution for a production environment is far more work than the original solution.

The full solution must be able to run at sites B, C, D, E and F, all of which have a different way of looking at the world. It must be fully documented. It must be full-featured. It must have a budget for support and enhancement. Version 2 of the may have to have some degree of backward compatibility.

None of this is new. We hold current HPC vendors to these standards. Successful open source projects meet similar requirements. But discussions about HPC Open Source System Software often have a flavor of "I know a piece of software that does that. Problem solved." More often than not, the software is far less robust than what would be supplied by a vendor.

To take a specific example, MPI is often considered a "solved" problem, with MPICH[7] as the solution. Another freely available implementation of MPI, LAM[8], has many features that make it an attractive candidate for clusters. MPICH is an excellent tool and research vehicle, but it would need a lot of work. For instance, its collective operations are untuned, it is not thread safe, it does not have a device appropriate for clusters in which processes are started and managed in a scalable way, it does not have robust multi-protocol support (e.g. for SMPs), it only barely works on some networks (e.g. Giganet). This is not to detract from the many excellent features of MPICH, such as the fact that source code is available, the ability to add new devices with minimal effort, the fact that the MPI efforts of many vendors were accelerated by MPICH, and that ongoing work will eventually add thread-safety and other MPI-2 features. Indeed, many people are using MPICH and LAM on small clusters today. The point is simply that a lot of work remains to be done, and it would be easy to underestimate what is needed.

2 Clusters are nice, but can they address the high end?

Clusters have for many years held out a promise of high performance computing at low cost. PVM was the first widely used software to make it easy to run parallel applications on a network of workstations. The NOW project and others started with the idea of harvesting unused desktop cycles, but eventually moved into the machine room as the political and technical difficulties became clear. Many other projects have made technical contributions.

The most visible of these projects is the Beowulf project, which has been quite successful in promoting the use of clusters. The most enduring contribution of the Beowulf project may be its recognition, early on, of the importance of open source software (before the term had been invented). The use of Linux rather than NT or Solaris or another operating system has made it possible to fix problems, extend the operating system to provide services needed for clusters, and has made it possible to build a cluster using almost any processor and almost any network.

For all of the attention, however, clusters seem to have been spinning their wheels for some time. Clusters are still largely a do-it-yourself affair, and what tools do exist are neither integrated nor scalable. Very recently, a number of companies have developed cluster administration tools. The tools generally do not interoperate, and each addresses a small part (often the same part) of the large picture. It is not clear whether any are designed for a high degree of scalability.

Much of the software being developed for clusters is open source, making it possible for hundreds or perhaps thousands of individuals and small organizations to put together small clusters. Nevertheless, the

[7] MPICH is pronounced "M-P-I-C-H" and is available from Argonne Lab at http://www.mcs.anl.gov/mpi/mpich.

[8] LAM is pronounced "LAM" and is available from the University of Notre Dame at http://www.mpi.nd.edu/lam/.

state of this software is primitive. Small pieces are relatively well developed, such as MPICH, LAM or PBS. But the glue needed to use these in a cluster is the software equivalent of duct tape, and almost no software is robust enough for what we would consider a production environment. More worrisome is that very little of the software being developed today is being designed with very large (10,000+ processor) machine in mind. This is not surprising, given that almost no developers have access or ever will have access to such machines. Nevertheless, it means that as cluster software becomes more mature, it is likely to "lock in" non-scalable interfaces

2.1 The need for coordination and integration of cluster system software: this is not an embarrassingly parallel problem.

The software needed to run a cluster is a superset of a standard Linux distribution. Cluster software is to first order a set of packages you can add to a standard Linux distribution.

Consider the following subsystems and partial descriptions of what they need to do.

- Cluster installation software needs to know how to interact with hardware at a low level (e.g. a MAC address associated with a certain Ethernet interface and a serial port number to access a console) and it needs to know what hardware is on the node. In some cases it can figure this out essentially automatically and in other cases it needs to be told. This software also needs to know how a node will be used – as a compute node, interactive login node, NFS server node, etc.

- A batch system package needs to know which nodes are compute nodes, something about their hardware, where server nodes are located. It also needs to know dynamic information, such as whether as node is up. It may know this because one of its daemons does not respond, but the package itself should not need to be an expert in deciding whether a node is up. It may need to organize its communication and server structure so that there is no bottleneck where congestion increases with the size of the system.

- A cluster-monitoring package needs to know about hardware configuration. It is an expert in deciding whether a node is up. It also has a lot of information that might be of interest to an accounting package, or a package that would like to predict when a disk might fail, or alert an operator when the temperature gets too high. The volume of information may be large, so it should collect this data in a scalable way to avoid congesting the network.

- A process manager needs to start up a large number of tasks very quickly. It may want to actually send binaries in a scalable way to avoid a server bottleneck. It collects standard output and error from the nodes, and as usual needs to do this in a scalable way.

- An MPI package will be concerned with at least three things. First it needs to know what network it should use. We should expect that MPI programs to not need to be relinked to run on clusters with different networks, and that MPI figures out what network to use. Second, it needs to know where to start processes. Third, it needs to be able to start up and manage its processes – in a scalable way, as usual.

It should be evident to the reader that just these five packages (and there would be many more) require much common functionality, and must interact closely with one another.

- Several packages use or store information about hardware and software configuration, or about runtime information such as whether a node is up or down.
- Several packages require scalable operations over arbitrary sets of nodes on the cluster.

An integrated system software package for clusters should probably include these two pieces as separate software components. But making this happen is not easy.

First, such components do not currently exist. Second, if they are to be used by a large number of other packages, they must have a very well defined interface that meets a large number of requirements and has a

critical mass of technical acceptance. Third, if the components are to be useful, developers must agree to use the interfaces they provide.

If the common functionality is not abstracted at the beginning, it may be very hard to replace it later. Furthermore, the common functionality may be quite complex and it is better to do it right once than wrong several times (think about security, for instance, in the context of multi-node operations).

It could be argued that this is a detailed technical issue that should be addressed by proposals, not in a roadmap document. The reason to consider it at this stage because it shows that the current "separate package" model of cluster software development probably won't work, it suggests a model of coordinated development that a lot of developers might not be comfortable with, and it could greatly affect the way any call for proposals was constructed. If there could be consensus on the technical side about the best way to coordinate development, this could make the management problem much easier.

Note that the interoperability problem does not apply to all software. For example, numerical packages typically rely on interfaces that are already well defined: programming languages and MPI.

2.2 The problem is primarily one of engineering and development, not research

The question of what is "research" is a touchy subject, because at the labs the word has a high status, because the it sometimes lets you get to the end of a project without doing the hard work of making your software useful, and because government funding is often part of a "research program". We make the following observations:

- We are proposing to put ourselves on the hook to produce a robust and very complex software system used by thousands of people to get their work done. If we commit to building scalable open source system software, it is not an option to come back in three years and say it can't be done.
- Distributed memory machines have been built at the level of several thousand processors. Quite a bit of research has been done in how to make them bigger. We can't say it's an unsolved problem. What we can say is that it is a huge development, testing and integration problem, one that is so large that we do not expect a single vendor to be able to solve it.
- Such engineering and development is directly relevant to research missions of the labs. There are countless examples of special purpose "research tools" in the lab that are not themselves the object of research, but enable research.

What we need to do is perhaps analogous to NASA putting together the space shuttle, or the physics community building a supercollider.

The reason all this is important is primarily because funding assumptions for development may be quite different from those for more speculative research. It is easy to lose focus from the primary requirement to build robust system software for large clusters.

All this is not to say that there is no research. First, there are certainly key items on the "critical path" that need to be understood. Second, while the near term goal may be to develop HPC infrastructure, in the longer term we will need software for Petaflop systems, which may or may not look anything like today's clusters. There are also many things not on the critical path that will move the HPC user environment beyond its current sorry state. The main point here is that this project is not business as usual nor is it our usual business.

2.3 Clusters should make life easier

Clusters currently have some catching up to do. They don't currently have the production features of other systems. We could justify this with "they don't cost as much either" or we could try not to go through another reduction in functionality for HPC. There is no question that clusters can catch up, and when they do catch up, the question will become how can we make them better than other systems. Everyone will have a laundry list here, but we can single out some items as neat things about clusters can do that other parallel machines would have trouble with. These include.

- Desktop to Petaflop integration. With the prevalence of Linux desktops, users should be able to develop on their own workstation, and send jobs transparently to a "back-end" supercomputer.
- Access to mass-market tools. With the explosion of commercial software for the Linux market, we have access to tools and functionality that were never available for the smaller Unix workstations market.
- If we solve the problem for large clusters, and we have been careful along the way, we have probably also solved the problem for small clusters. Everything we do on a very large scale should also make it easier for Joe to put together a personal supercomputer from discarded Itanium processors.

3 The Promise of Open Source

3.1 Why Open Source is a Good Thing

The last few years have seen an explosion of interest in what is now called Open Source technology – software for which source code is available without charge for any purpose, modifiable, and redistributable. Tens of billions of dollars have been invested in the belief, once rare, that Open Source can be a central part of a viable business model[9].

Open source software has a number of advantages over traditional software in the context of scientific research. It facilitates technical innovation and reduces development costs because developers can start with a robust software base, rather than starting from scratch. It encourages good software design because it enables scrutiny. It promotes good science because it allows reproducibility, peer review, and extension of research into new directions. We do not assert that the quality of open source software is necessarily better than other software, or that open source magically creates additional manpower.

Two features of open source make it appealing and indeed necessary for HPC system software.

First, open source reduces the risks of an unstable HPC marketplace. Relying on a single vendor for non open-source software is a very dangerous proposition. The vendor could go out of business; the vendor could decide not to support your new hardware. A particularly sad example of this was when Thinking Machines' software, notably CMSSL, CM Fortran and Prism, disappeared into a black hole for several years when TM went under. It has since resurfaced as a Sun product, but all of the momentum and intellectual investment from users were lost. This is not to say that Open Source would have saved these particular products; but it is true that in the Open Source world it is common for software developers/maintainers to find something else to do, for the software to sit for a short while, and then for someone else to pick it back up, improving and maintaining it.

Second, open source is a necessary ingredient for building critical mass behind software with numerous small contributors. Current HPC software efforts are fragmented. For instance, most vendors have proprietary "solutions" for fast parallel file systems, but none are very good; much system software is tied to proprietary operating systems, forcing reinvention of the same concepts many times. There exist open source tools that could be developed into production software, but the resources of the HPC community are squandered because we pay vendors to do the same development many times and because development in the academic and laboratories is uncoordinated. Consider, for instance, the development of MPI. Most vendors started with a government-funded implementation of MPI. Through computer procurements and laboratory R&D, the government has several times funded improved implementations. But improvements in one version don't appear in other versions, and the developers and maintainers of the original open source code struggle for funding.

What is needed to break the cycle is coordinated funding of development in which open source is a prerequisite for participation.

[9] It is important to keep in mind that commercial interest is no proof in the advantages of open source. Its only relevance to the arguments presented here is that the open source infrastructure is quite robust and become more so with increased investment.

It is reasonable to ask whether "source available" [10] is as good as open source. The answer is no. While having access to non-open source code may make it possible to fix some bugs, the efforts of the community are fragmented. A bug fix from site A cannot be distributed to site B. Moreover, software developed for computer X cannot be applied to computer Y. For the government[11] to fund modifications to proprietary HPC software without guaranteeing that the modifications will be available is foolhardy, as the government is the primary customer and has to pay for the same item several times.

The benefits of Open Source software derive from meeting all of the Open Source definition. A great deal of thought has gone into the definition, and if any of the conditions aren't met, the benefits of Open Source disappear. The white paper by Beckman and Oldehoeft[12] contains a thorough introduction to Open Source concepts and lays out in much more detail why Open Source is good for science.

3.2 Why the open source requirement must be firm

The reason the current untenable state of HPC system software is that its development is splintered. Overall, a great deal of effort is being put into developing HPC system software, but this effort is not producing robust highly scalable systems. The solution we propose here is to focus and expand these efforts to develop a single unified infrastructure. Any part of that infrastructure that is not open source becomes a potential roadblock to progress. We suggest that any new initiative on HPC Open Source System Software take a firm position[13] so that we don't find ourselves in three years in the same position we're in today.

There has been some discussion of a middle ground, where cluster software could be shared by certain key players but not freely given to others. It is the strongly held opinion of this author and many others in the HPC OS WG that this would be a big mistake. It would cut us off from a huge pool of testers and contributors, make vendor participation difficult, as well as make it necessary to have two completely different software bases – one for very large systems in the labs, and one for smaller systems for everyone else.

Any coordinated effort in system software development will need to address source code licensing up front, establishing uniform standards and policies for source code. In the current ad hoc system, some labs/agencies have been successful at distributing source code, and others have not. Even between DOE labs, there are currently significant barriers to the code-sharing that is necessary for cooperation. It is difficult to determine to what extent these barriers are imposed by lawyers and to what extent they are facilitated by developers. Uniform standards and policies will make life easier for everyone.

The issues of export control and U.S. technological superiority always arises in a discussion of open source software for cluster computing. This is the topic of another paper, but the short summary is that this is a non-issue in the opinion of many. Someone should write a white paper on this topic so that the arguments so not need to be reformulated and repeated each time it comes up.

There are two general areas where non-Open Source software might be acceptable.

The first area is software used by the mass market, where HPC does add additional requirements. The mass market will ensure that HPC needs are met. The best example is compilers.

---

[10] By "source available," we mean that source code is available but may not be modifiable and/or redistributable.

[11] The "government" is ultimately the largest consumer of HPC software in the U.S. In this context, the government includes not only labs, but also academic researchers (who are usually funded by the government). We caution against an interpretation that software can be made available to the "government" but restricted for other use. This would be huge mistake that would cut off the community in many ways.

[12] *The Case for Open Source Software Development for Scalable High Performance Computing*. Pete Beckman and Rod Oldehoeft. LANL report number LA-UR-00643. February 2000.

[13] How to take a "firm position" is not clear. There are limits to what any government program could require.

The second example is where there is a well-defined and agreed-upon interface in the cluster infrastructure, where a non open-source module[14] could be plugged in. The only way you know for sure that the interface is well defined is if there are two solutions that can be plugged in. These might both be proprietary, or one might be the bare-bones open source version and the other a featureful proprietary version. One example might be a debugger that relies on interfaces for X, symbol table formats, internal MPI message queue structure, and so forth. The risk is that even if this is technically feasible, it tends to lock in the interfaces so that they cannot evolve.

3.3 What about NT?

There are two reasons why NT development should be discouraged in this HPC-OS context.
The first is of course that NT is not open source, and suffers from the generic arguments in previous sections. Ultimately it boils down to this: the HPC community must be able to be self-sufficient. NT, primarily because it is open source, does not provide the control needed.

The second reason is the integration issue we have stressed in this paper. Software written to work in an NT environment will generally not work in a Linux environment, so NT and Linux software will go their separate ways. To have integration you have to pick a framework and stick with it.

This is not to say that NT has no part in high performance computing. For instance, in the eyes of many people, the Windows development environment is far ahead of the Linux environment. One can imagine developing software to allow an NT workstation to interface seamlessly with a Linux-based cluster. The integration would not be easy, but this interaction is likely to be less complex than the interaction between components of cluster system software.

Linux vs. BSD would be another question entirely. Only the second applies, and it applies less strongly, since Linux and BSD have very similar interfaces. If you pick one, you have to pick Linux, not for technical reasons but for the practical reason that there are so many more eyeballs looking at it. One can imagine (though we would not advocate) a dual-OS approach (cluster software should run on both OSs) as a risk-mitigation strategy (e.g. Linus gets hit by a bus and all hell breaks loose).

3.4 Coordination and Integration of Open Source Software

A question central to any discussion of distributed development of integrated software is how a loosely organized group of developers can produce a highly complex and integrated software system, such as a full Linux distribution (of which the kernel is just a small part). Successful software companies such as Microsoft have a management structure that ultimately controls every part of the development process. Certainly open source software requires similar mechanisms to manage complexity?

There are three answers to this question. First, a Linux distribution comprises hundreds of relatively small, unrelated pieces. These pieces fit together because they rely on well-defined interfaces such as Posix, the X window system, the Linux kernel device driver interface, the RPM specfile format or any number of other interfaces, both formal and informal. Without these interfaces, software would have to be monolithic. Moreover, each identifiable piece is developed primarily by a small team. Hundreds may participate in testing and debugging, but there is always a small core of developers.

Second, in several essential areas, in fact an organizational structure keeps things under control. This is most visible in the development of the kernel itself, where Linus and a handful of others control what goes into the kernel. This core group in turn relies on a set of second-level core developers. Such organization is also important, for instance, in the Gnome project and anywhere that interfaces and specifications are evolving.

[14] Such a non-open source module would probably have to be provided in source form, so that it could be compiled. Such software should not hold back progress in other areas, which a binary-only distribution might do.

Third, it can be argued that the most sophisticated and polished software is application software such as Netscape Navigator, Star Office, and others. Even when these are open source, they are developed by an organized team with commercial backing.

The point of this argument, which will become important in the context of HPC, is that to develop a complex infrastructure with many inter-operating components, it is not sufficient simply to throw money at a number of development groups, tell them to write open source software, and hope that something good will happen. Coordination must be carefully considered and included as part of an overall strategy.

4 Open source for other HPC software – applications, middleware, and so forth.

High Performance Computing systems are of course no use by themselves. Layers of software that run on top of the operating system are equally important. These layers of software are outside the scope of this paper. In particular, we have not identified a looming crisis that may prevent continued progress three years from now.

However, these other areas may help to inform the discussion of cluster software. While coordinated system software development is relatively rare, there exist several examples in other areas where a community decided to contribute to a common infrastructure. One example would be the Globus project, which is centered at ANL and ISI, but which has gathered the support of researchers at many labs who are contributing software that works with the Globus framework. Another example would be community codes that are being used by the so-called Grand Challenge projects in DOE.

5 Towards a high-end open source solution

We hope that the reader has been convinced the HPC community needs to take responsibility for its own survival in a world where it has less and less economic clout. We hope furthermore that the reader has been convinced that this requires an investment in HPC Open Source system software. The next question, then, is how do we go about doing this.

Before continuing, a caution: keep in mind that old saying that you should be careful what you ask for – you might get it. One possible outcome of this whole exercise is that some months from now some folks in Washington decide that all this is a good idea, and send resources to “make it so”. The scope of the project we are talking about is unprecedented. The government has no track record of successfully producing large complex software systems developed in a distributed fashion. To the extent that a track record exists, it is not wonderful.

The first and most important step is to identify requirements and assess current technology. This forms the basis for a roadmap that defines the starting point, the end point, and describes possible development paths. This is exactly what is being addressed by the HPC Open Source Working Group.

As has been argued above, however, it is not sufficient to identify technology areas and throw money at the problem – the process must be structured to ensure that software is scalable and integrated.

The second step is therefore to explore how a coordinated approach to HPC Open Source might proceed.

5.1 The Roles of Government and Industry

Discussion of how government and industry might interact in this project has been characterized more by “warm fuzzies” than by a concrete model of interaction.

Recall:

- The HPC community cannot rely on the market and should take the initiative to solve its own problems. The HPC community is primarily government users.
- Traditionally, high performance computing systems have been procured from vendors who supply complete hardware/software solutions.

- A major contributor to the anticipated HPC "crisis" is that no single vendor can marshal the resources to produce a complete solution to the problem.

5.1.1 The stone soup model.

To first order, our model is that everyone should write open source software, contributing to a single software base that is owned by the community as a whole. In this model, everyone contributes, and everyone benefits. A lab puts its finite resources into a great process management package, which goes in the pot. A company makes its name in parallel file systems. Another company makes a great local filesystem. Everyone puts in a small amount and gets a lot. If done right, everyone has better software than if they tried to do it themselves.

This is a warm fuzzy. At least two questions should be resolved. 1) How can vendors distinguish themselves if they are all using the same software base, and 2) How do we make sure everything works together and meets the requirements.

We don't see a clear answer to these questions, but present some ideas below.

5.1.2 The "reference implementation" model

One model for interaction with industry is that this open source software base is a "reference implementation" that may be used by vendors as a starting point for their own proprietary or enhanced systems. We believe it would be a mistake to encourage[15] this model. The advantages of open source derive from contributions back to the community, and fragmentation is what got us here in the first place. See the discussion in the open source section.

Although we dismiss this approach, we note that it has had some notable successes in the past, but that these successes don't apply to our current predicament. Both PVM and MPI were implemented first under government funding and then widely implemented by vendors based on the government-funded "reference implementations." While these vendor implementations helped get vendor HPC environments off the ground, the main effect was probably to accelerate the timetable rather than enabling new functionality. In the case of PVM, the lack of a very clearly defined standard and a model not well-matched to most supercomputers also led to lack of portability that made PVM difficult to support on vendor systems.

5.1.3 The government-does-the-research, industry-does-the-implementation model

This is closest to the Silicon Industry Association model. It doesn't make a lot of sense to us because there are not large unanswered research questions where the labs are far ahead of industry. The truly unanswered research questions are in the area of Petaflop and other exotic systems, and if the government focuses on that area, we have not solved any of our immediate problems.

5.1.4 Labs and vendors form explicit partnerships

This approach defers discussion of government-industry partnership to the proposal process. This may be the right approach, but ultimately all of the same issues will have to be addressed.

5.1.5 Fund industry to do new open source development

This model is being used by LANL.

---

[15] Note the important distinction between encouraging a way of doing things and requiring it. In many cases, we may not be able to impose restrictions (e.g. companies are free to do what they want with software they develop, and certain licenses impose restrictions).

5.2 How vendors can stay happy

The stone soup model is a utopian ideal, but we have identified two problems. We address here the issue of vendor competitiveness.

If vendors contribute their software to the Open Source pool, how can they stay competitive? There may still be many ways for vendors to distinguish themselves. These include:

- A strong record of contributing to the Open Source infrastructure gives companies more credibility in the market. This may be the Red Hat business model, but we're not sure.
- Some software will necessarily be hardware-specific. For instance, software to power nodes on and off or report hardware status information will be quite different for different hardware. Vendors can make sure that the software for their hardware is very good. Note that this does not mean separate software packages, but separate modules, which plug into a controller as components. For instance, a cluster-controller package needs the ability to power-cycle nodes and to monitor node status. Vendor X, with fancy node-monitoring hardware, can build an Open Source cluster controller that has all sorts of bells and whistles on the fancy hardware, and displays "not implemented" for less fancy hardware.
- Cluster expertise will be important to buyers of clusters. Cluster expertise is gained largely by experience working closely with cluster software. We would rather buy a machine from a vendor that has written several critical subsystems than from one with no one on staff who is intimately familiar with them.
- Vendors can provide different levels of service. Vendor A sells you the hardware at very slim profit margin. Vendor B sells a full maintenance and support package to the discriminating buyer who realizes he will need to train and pay two sysadmins.
- Vendors can provide different levels of hardware integration. Vendor A may provide x86 nodes with off-brand memory plus fast Ethernet on shelves. Vendor B has figured out how to rack the systems nicely, knows what serial controller is best supported, has measured performance and can tell you what it will be, etc.
- Understanding and playing in the HPC arena may be a good investment for vendors. There is still plenty of opportunity for applying HPC technology to commercial applications, and being 6 months in front of the competition can be important.

Ultimately, however, no amount of rationalizing can make the competitiveness issue go away completely. There is another way to make vendors happy, which is to make sure that they are paid for their software development. If we buy computers based on the robustness of the software environment and require that the environment be open source, ultimately the question of competitiveness will arise. Rather than asking vendors to ante up software development costs, we should explicitly fund Open Source development. Often it will make the most sense to fund this development through private industry, rather than through the labs.

5.3 How to coordinate and achieve integration

This is a touchy subject because no one is interested in top-down management and being directed by Washington or even a technically competent benevolent dictator. How can we achieve integration without a heavy hand?

The process obviously starts (on the government side, that is) with the funding process. The spirit in which these ideas are suggested is that not that this is the way things must be, but that we perceive that the "normal" review criteria aren't up to the task. Here are some ideas for variations on the "normal" process.

- One possibility is to solicit very large proposals that integrate many pieces of software. This is better for coordination but worse for assigning responsibility.
- Projects should be required to produce robust software that meets community needs. How do we determine community needs and whether projects are responsive? Peer proposal review peer mid-review and peer post-review are a good start, with the twist that "peers" include (perhaps exclusively)

users of the software, which will be other developers getting funding from the same source as well as vendors. In a normal academic peer review, reviewers are financially disinterested. Basically the project should answer to customers. Peers may also direct the work, as they see fit.

- Proposals should identify precisely where the proposed software interacts with other software and give an integration plan that is checked up on later by technically knowledgeable people. This goes well beyond normal "related work" discussion.
- Continuation of multi-year projects should depend strongly on deliverables, and deliverables should be robust implementations with limited functionality rather than prototype implementations with grandiose plans.

Another way to say this is that coordination should not be expected to happen spontaneously. It should be an important part of the proposal process.

One can also conceive of a technical committee to help guide the integration process. Some possible roles of such a committee/committees are:

- Determine standards, a la IETF (not MPI), which requires working implementations and is not so much a "design by committee" as our own MPI and HPF standards. (The MPI/HPF process is good, but perhaps less appropriate to system software).
- Resolve disputes.
- Facilitate discussion (thereby preventing disputes)

Acknowledgements
This work was supported by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC03-76SF00098.